Department I - C Plus Plus

Modern and Lucid C++ for Professional Programmers

Week 2 – Functions, Values and Streams

Prof. Peter Sommerlad / Thomas Corbat
Rapperswil, 24.09.2019
HS2019

Cevelop
Your C++ deserves it

HSR
HOCHSCHULE FÜR TECHNIK
RAPPERSWIL
FHO Fachhochschule Ostschweiz

IFS INSTITUTE FOR SOFTWARE

Recap Week 1

Cevelop
Your C++ deserves it

HSR
HOCHSCHULE FÜR TECHNIK
RAPPERSWIL
FHO Fachhochschule Ostschweiz

IFS INSTITUTE FOR SOFTWARE

- **What is the output?**

Point.h

```
struct Point {
  int x;
  int y;
};
inline Point modify(Point in) {
  in.x = 3;
  in.y = 4;
  return in;
}
```

main.cpp

```
#include "Point.h"
#include <iostream>
int main() {
  Point point{1, 2};
  Point other = modify(point);
  std::cout << '{' << point.x << '/' << point.y << "}\n";
  std::cout << '{' << other.x << '/' << other.y << "}\n";
}
```

- **How do you prevent the violation of the One Definition Rule?**

Point.h

```cpp
struct Point {
  int x;
  int y;
};
```

Square.h

```cpp
#include "Point.h"
struct Square {
  Point p1;
  Point p2;
};
```

main.cpp

```cpp
#include "Square.h"
#include "Point.h"
int main() {
  Square s{Point{1, 2}, Point{9, 9}};
  //...
}
```

Topics

- **Variable Definitions**
- **Values and Expressions**
- **Strings and Sequences**
- **Formatted I/O**

Variable Definitions

Cevelop
Your C++ deserves it

Goals:

- You know how and where to define variables.
- You want to make all your variables const

HSR
HOCHSCHULE FÜR TECHNIK
RAPPERSWIL
FHO Fachhochschule Ostschweiz

IFS INSTITUTE FOR SOFTWARE

```
<type> <variable-name>{<initial-value>};
```

Examples

```
int anAnswer{42};
int const zero{};
```

- **Defining a variable consists of determining its `<type>`, its `<variable-name>` and its `<initial value>`**
- **Initialization might be omitted for non-const variables, but that is bad practice and potentially dangerous**

```
double x;
```

- **Empty braces mean default initialization**
- **Using = for initialization we can have the compiler determine its type (do not combine with braces!)**

```
auto const i = 5;
```

```
int const theAnswer{42};
```

- **Adding the const keyword in front of the name makes the variable a single-assignment variable, aka a constant**
  - A `const` variable must be initialized
  - A `const` variable is immutable

```
int const theAnswer;
```

```
theAnswer = 15;
```

```
theAnswer++;
```

- **Some constants are required to be fixed at compile time**
  - To enforce that use the keyword `constexpr`
  - We will learn more about that in C++ Advanced

```
double constexpr pi{3.14159};
```

- **The keyword const also appears in other contexts**
  - It always denotes something immutable (there is also a `mutable` keyword).

You should use `const` whenever possible for non-member variables!

- **Why shoud I use const?**
  - A lot of code needs names for values, but often does not intend to change it
  - It helps to avoid reusing the same variable for different purposes (code smell)
  - It creates safer code, because a const variable cannot be inadvertently changed
  - It makes reasoning about code easier
  - Constness is checked by the compiler
  - It improves optimization and parallelization (shared mutable state is dangerous)
- **Computing values and functions only without side-effects is possible and a specific style supported by C++**

Where to Place Variable Definitions?

- **As close to its use/as late as possible**
  - Do not practice to define all (potentially) needed variables up front (that style is long obsolete!)
  - More chances for "const"-ness: single assignment
- **Scoping rules are similar to Java:**
  - A variable defined within a block is invisible after the block ends
  - Difference: Avoid name clashes, i.e., redefining an existing variable inside a block is not an error in C++
- **Every mutable global variable you define is a design error!**
  - Code using (non-const) globals is almost untestable
  - Concurrent code with globals requires careful synchronization!
  - The Ctylechecker plug-in will warn you if you do this

A Note on Naming

- **The C++ convention is to begin variable names with a lower case letter**
- **Spell out what the variable is for**
- **Do not abbreviate `uncsrly`**

```cpp
int const mpm = 1609;
```

```cpp
int const metersPerMile = 1609;
```

- **Very short (one letter) names can be used in tightly bound context, e.g., for iteration indices or very short scopes**

```cpp
for (auto i = 0; i < size; i++) {
  //...
}
```

Types for Variables

- **C++ has a whole bunch of built-in types, mostly for numbers**
  - They are part of the language and don't need an `#include`
  - **`short, int, long, long long –`** each also available as **`unsigned`** version
  - **`bool, char, unsigned char, signed char`**
    - They are treated as integral numbers as well
  - **`float, double, long double`**
  - **`void`** is special, it is the type with no values
  - Plus some more not relevant now
- **The standard library provides a multitude of types for different purposes (defined in classes)**
  - Important: **`std::string`** and **`std::vector`**
  - Their use requires **`#include`** of the type definition

Values and Expressions

Goals

- You can identify the type of a literal
- You know the most important operators

Cevelop
Your C++ deserves it

HSR
HOCHSCHULE FÜR TECHNIK
RAPPERSWIL
FHO Fachhochschule Ostschweiz

IFS INSTITUTE FOR SOFTWARE

Literal Value Examples

Literal Example	Type	Value

`'a'`
`'\n'`
`'\x0a'`		

`1`
`42L`
`5LL`
`int{} (not really a literal)`		

`1u`
`42ul`
`5ull`		

`020`
`0x1f`
`0XFULL`		

`0.f`
`.33`
`1e9`
`42.E-12L`
`.3l`		

`"hello"` `"\012\n\\"`		

Literal Value Examples

Literal Example	Type	Value
'a'	char	Letter a, value: 97
'\n'	char	<NL> character, value: 10
'\x0a'	char	<NL> character, value: 10
1	int	1
42L	long	42
5LL	long long	5
int{} (not really a literal)	int	0 (default value)
1u	unsigned int	1
42ul	unsigned long	42
5ull	unsigned long long	5
020	int	16 (octal 20)
0x1f	int	31 (hex 1F)
0XFULL	unsigned long long	15 (hex F)
0.f	float	0
.33	double	0.33
1e9	double	1000000000 (10^9)
42.E-12L	long double	0.000000000042 ($42*10^{-12}$)
.3l	long double	0.3
"hello"	char const [6]	Array of 6 chars: h e l l o <NUL>
"\012\n\\"	char const [4]	Array of 4 chars: <NL> <NL> \ <NUL>

- **Arithmetic**
  - binary: `+ - * / %`(modulo)
  - unary: `+ - ++ --`
- **Logic**
  - ternary/conditional: `?:`
  - binary: `&& and || or`
  - unary: `! not`
- **Bit-operators**
  - binary: `& | ^ << >> bitand bitor xor`
  - unary: `~ compl`
  - Use unsigned types of bit operators

What are the values?

```
1. (5 + 10 * 3 – 7 / 2)
2. auto x = 3 / 2;
3. auto y = x % 2 ? 1 : 0;
```

1. **Precedence as in normal mathematics**
   - `5 + 30 - 3 => 32`
2. **Fraction results of integer operations are always rounded down**
   - `3 / 2 => 1`
3. **Integer to boolean conversion**
   `0 -> false` / every other value -> `true`
   - `true ? 1 : 0 => 1`

Caution: Type Conversion

- **C++ provides automatic type conversion if values of different types are combined in an expression**
  - Unless in braced initialization

```
int i{1.0};
```

- **Division of integers does not round**
  - **`double x = 45 / 8;`**

Value of `x: 5`

- **Dividing integers by zero is undefined behavior**
  - that is also true, when using modulo **`(5%0)`**

Undefined Behavior

left: lvalue `x = 6 * 7;` right: rvalue

- **Assignment requires a variable on the left side: an lvalue (x)**
  - Elements in a container can also act as an lvalue
- **The value on the right side is an rvalue: `6 * 7 = 42`**
- **Most binary operators can be combined with assignment to shorten the code**
  - `a += b; c /= d; x >>= 2;`
- **Increment/Decrement require an lvalue**
  - `a++;`
  - `++b;`
  - `5++;`

- **Relational Operators compare values**
  - `< > <= >= == !=`
  - results in `true` or `false`
- **Logical operators and conditional statements are generous to accept numeric values as statement of truth**

```
if (5);
while (1);
std::cout << (!x % 2 ? "even" : "odd ");
if (a < b < c);
```

Compiles, but what does it mean?

- **Use `double` - usually most efficient on current hardware and default for floating point literals**
  - Use `float` only, if memory consumption is utmost priority (very very large data sets) and precision and range can be traded (on 64bit often not beneficial)
- **Remember there are legal double values that are not numbers:**
  `NaN`, `+Inf`, `-Inf` (not-a-number, plus/minus infinity)
- **Comparing floating points for equality (==) is usually wrong**
  - CUTE's `ASSERT_EQUAL(expected, actual)` automatically provides a “delta" value as a margin to consider almost equal values equal
  - Or use `ASSERT_EQUAL_DELTA(expected, actual, delta)`

Strings and Sequences

Goals:

- You know the basic sequence containers std::string and std::vector
- You are aware of the unspecified sequence of argument evaluation

Your C++ deserves it

HSR HOCHSCHULE FÜR TECHNIK RAPPERSWIL

FHO Fachhochschule Ostschweiz

INSTITUTE FOR SOFTWARE

- **`std::string` is C++'s type for representing sequences of char (which is often only 8 bit)**
  - Unicode support is different from Java (Advanced C++)
  - Literals like `"ab"` are not of type `std::string`
  - But `"ab"s` is (requires `using namespace std::literals;`)

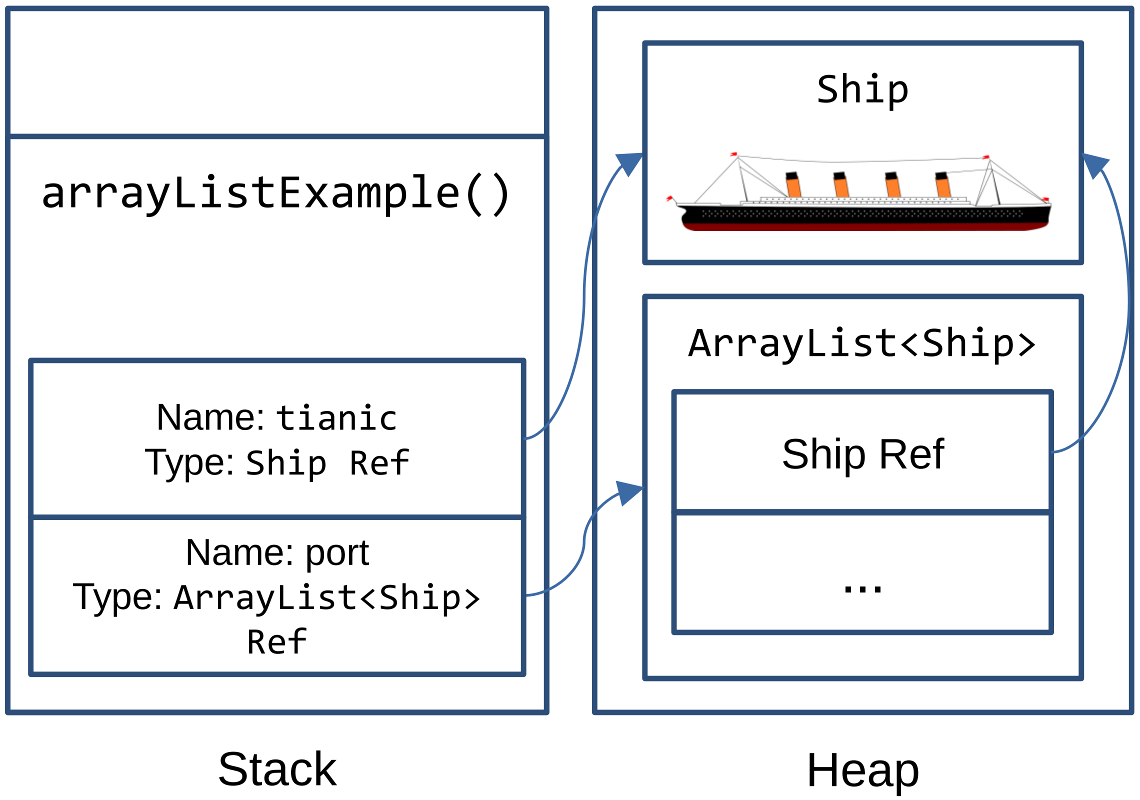
```
std::string name{"Carl"};
```

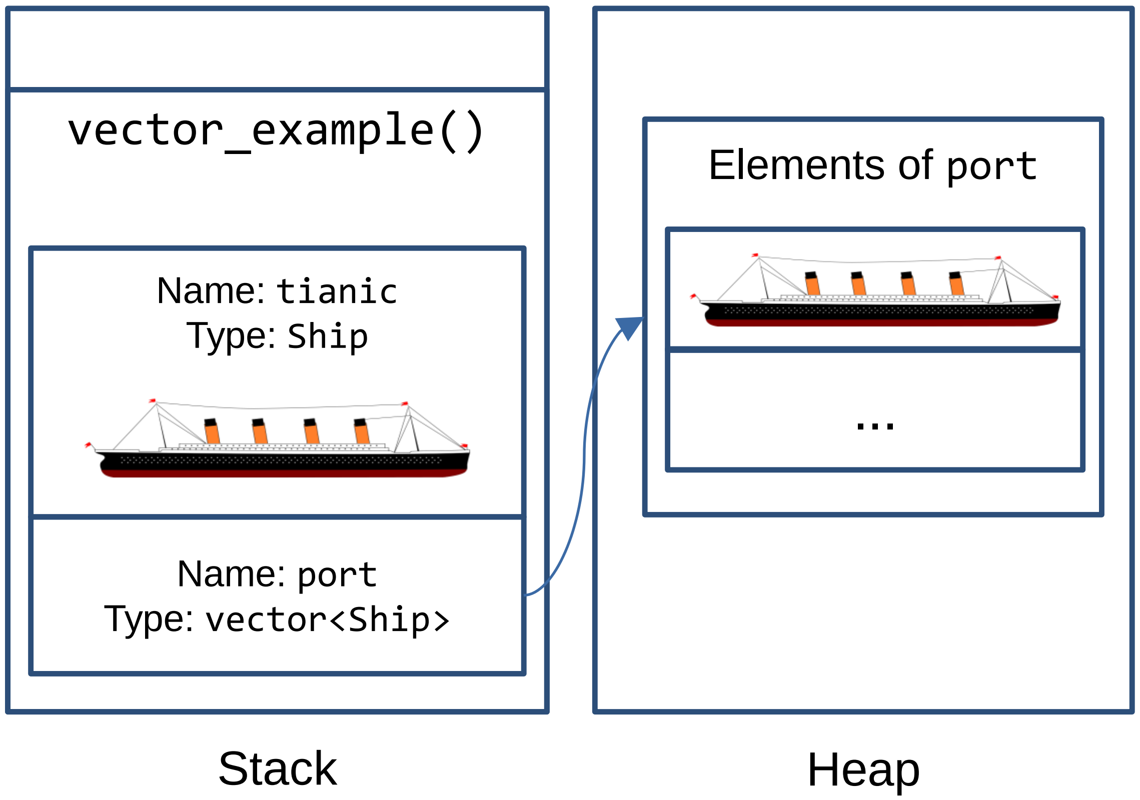
- **`std::vector<T>` is a homogeneous container representing a sequence of values of type `T`**
- **Almost all types can be a vector element `T`**
  - No need to reserve space for individual elements! (Container != Collection)
  - C++ vs Java: Contains copies of the elements not references

ArrayList<E> vs std::vector<T>


```java
public class SomeClassWeDontReallyNeed {
  public static void arrayListExample() {
    Ship titanic = new Ship("RMS Titanic");
    ArrayList<Ship> port = new ArrayList<>();
    port.add(titanic);
  }
}
```

```cpp
#include "Ship.h"
#include <vector>
void vector_example() {
  Ship titanic{"RMS Titanic"};
  std::vector<Ship> port{};
  port.push_back(titanic);
}
```





Working with `std::string`


```cpp
#include <iostream>
#include <string>

void askForName(std::ostream & out) {
  out << "What is your name? ";
}

std::string inputName(std::istream & in) {
  std::string name{};
  in >> name;
  return name;
}

void sayGreeting(std::ostream & out, std::string name) {
  out << "Hello " << name << ", how are you?\n";
}

int main() {
  askForName(std::cout);
  sayGreeting(std::cout, inputName(std::cin));
}
```

```
void askForName(std::ostream & out)
```

- **`std::string` and built-in types represent values**
  - Can be copied and passed-by-value
  - No need to allocate memory explicitly for storing the chars
- **Some objects aren't values, because they can not be copied:**
  - Streams representing the program's I/O
- **Functions taking a stream object must take it as a reference, because they provide a side-effect to the stream (i.e., output characters)**
- **Reference parameters are marked with `'&'` (ampersand)**
- **In Java all objects are passed as references! (not the same kind of references as in C++!)**
  - Same name, different concept

Caution: Call Sequence

- **Statements are sequenced by ; (semicolon)**
- **Within a single expression, such as a function call, sequence of evaluation is undefined! (except for the comma operator ,)**

```
void sayGreeting(std::ostream & out,
                 std::string name1,
                 std::string name2){
  out << "Hello " << name1 << ", do you love " << name2 << "?\n";
}

int main() {
  askForName(std::cout);
  sayGreeting(std::cout,
              inputName(std::cin),
              inputName(std::cin));
}
```

Formatted I/O

Goals:

- You know how to read and write from and to streams
- You know about the possible states of an `std::istream`
- You can read input from an `std::istream` safely

- **Stream objects provide C++'s I/O mechanism**
  - Pre-defined globals: `std::cin` `std::cout` 😟
- **Use them ONLY in the `main()` function!**
- **"shift" operators read into variables or write values**
  - `std::cin >> x; std::cout << x;`
- **Multiple values can be streamed at once**
  - `std::cout << "the value is " << x << '\n';`
- **Streams have a state that denotes if I/O was successful or not**
  - Only `.good()` streams actually do I/O
  - You need to `.clear()` the state in case of an error

Reading a `std::string` Value


```cpp
#include <iostream>
#include <string>

std::string inputName(std::istream & in) {
  std::string name{};
  in >> name;
  return name;
}
```

- **Reading a `std::string` can not go wrong, unless the stream is already `!good()`**
  - The content of the `std::string` is replaced
  - Maybe the `std::string` is empty after reading

Reading an `int` Value

```
int inputAge(std::istream& in) {
  int age{-1};
  if (in >> age) {
    return age;
  }
  return -1;
}
```

- No error recovery
- One wrong input puts the stream into status fail
- Characters remain in input

Caution: Cevelop and EOF


```cpp
#include <iostream>

int main() {
  size_t count{0};
  char c{};
  while (std::cin >> c) ++count;
  std::cout << count << "\n";
}
```

```
$ mycharcount < input.txt
42
$ mycharcount
12345
<CTRL-D>
6
$
```

- **If you write programs to read all of the input you need to terminate the input:**
  - Ctrl-D (Linux/Mac) and Ctrl-Z (Windows)
  - You might need to (re)set the focus to the Cevelop console
- **Press <Enter> to send the current line to the input**
  - You may edit the line before sending, e.g. delete characters

Robust Reading an `int` Value

```
int inputAge(std::istream & in) {
  std::string line{};
  while (getline(in, line)) {
    std::istringstream is{line};
    int age{-1};
    if (is >> age) {
      return age;
    }
  }
  return -1;
}
```

- **Read a line and parse it as an integer until `OK` or `EOF`**
- **Read operation in while condition acts as a "did the read work?" check**
- **Use an `std::istringstream` as intermediate stream**

```
int readFrom(std::istream & is) {
  //...
}
```

State Bit Set	Query	Entered
<none>	is.good()	initial is.clear()
failbit	is.fail()	formatted input failed
eofbit	is.eof()	trying to read at end of input
badbit	is.bad()	unrecoverable I/O error

- **Formatted input on stream `is` must check for `is.fail()` and `is.bad()`**
  - If failed, `is.clear()` the stream and consume invalid input characters before continue

```
int inputAge(std::istream & in) {
  while (in.good()) {
    int age{-1};
    if (in >> age) {
      return age;
    }
    in.clear();  // remove fail flag
    in.ignore(); // one char
    // alt: in.ignore(std::numeric_limits<std::streamsize>::max(), '\n');
    // ignores whole line
  }
  return -1;
}
```

```
#include <iostream>
#include <iomanip>
```

I/O Manipulators:
`setw(n), setprecision(n)`

```
int main() {
  std::cout << 42 << '\t'
            << std::oct << 42 << '\t'
            << std::hex << 42 << '\n';
  std::cout << 42 << '\t' // std::hex is sticky
            << std::dec << 42 << '\n';
  std::cout << std::setw(10) << 42
            << std::left << std::setw(5)<< 43 << "*\n";
  std::cout << std::setw(10) << "hallo"<<"*\n";

  double const pi{std::acos(0.5) * 3};
  std::cout << std::setprecision(4) << pi << '\n';
  std::cout << std::scientific << pi <<  '\n';
  std::cout << std::fixed << pi * 1e6 <<  '\n';
}
```

```cpp
#include <iostream>
#include <cctype>
int main() {
  char c{};
  while(std::cin.get(c)) {
    std::cout.put(std::tolower(c));
  }
}
```

- **A very simple program transforming its input to lower case**
  - `<cctype>` contains character conversion and character kind query functions (`std::tolower(c)`, `std::isupper(c)`)
- **`get()` and `put()` are unformatted I/O functions**
  - What happens when we use `>>` and `<<` ?
- **More in the exercises for you to experiment with!**

- **All values have a type**
- **rvalues have only value, lvalues also a location (=variable)**
- **Variables can keep a value and thus also have a type**
- **`const` makes variables single-assignment only, no changes**
- **Output can be done using `ostream`, i.e., `std::cout` and `<<`**
- **Input uses `istream`, i.e., `std::cin` and `>>` to an lvalue**
- **Streams have a state for `eof` and format errors on input**